Training Deeper GNNs

Prof. O-Joun Lee

Dept. of Artificial Intelligence,
The Catholic University of Korea
ojlee@catholic.ac.kr

네트워크 과학 연구실
NETWORK SCIENCE LAB

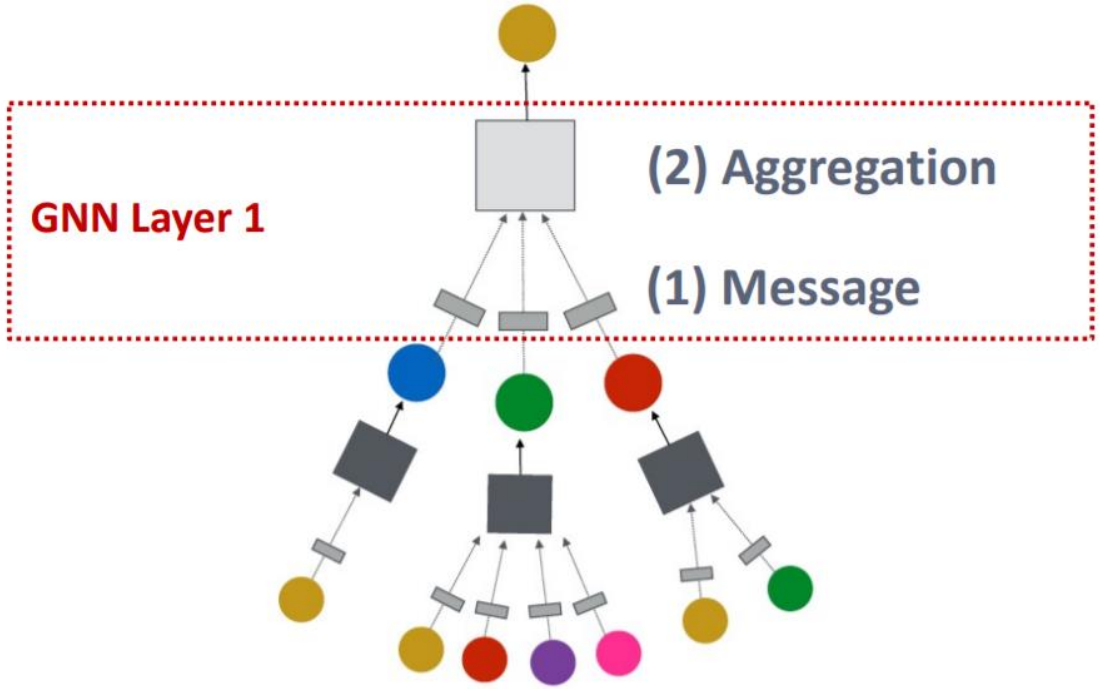
가톨릭대학교
THE CATHOLIC UNIVERSITY OF KOREA

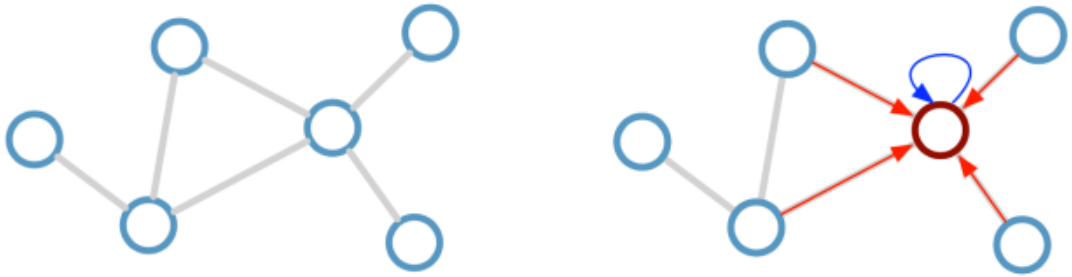
Contents

- Why do we need deeper GNNs?
- What impedes GNNs to go deeper?
- Solutions for alleviating over-smoothing and over-fitting
  - Dropping Nodes and Edges
  - JK networks with Skip Connections
  - DeeperGCN with residual connection
  - GCNII with residual connection and identity mapping

Recap: Graph Convolutional Networks (GCNs)

- GNN Layer = **Message** + **Aggregation**
  - Message COMPUTATION
    - how to make each neighborhood node as embedding?
  - Message AGGERGATION
    - how to combine those embeddings?





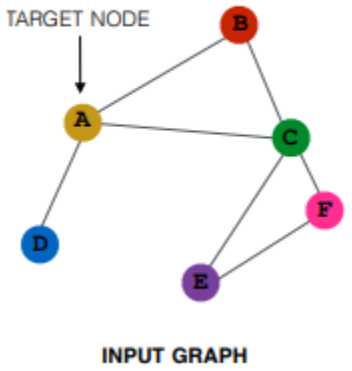
Update rule: $$\mathbf{h}_i^{(l+1)} = \sigma\left(\mathbf{h}_i^{(l)}\mathbf{W}_0^{(l)} + \sum_{j\in\mathcal{N}_i}\frac{1}{c_{ij}}\mathbf{h}_j^{(l)}\mathbf{W}_1^{(l)}\right)$$

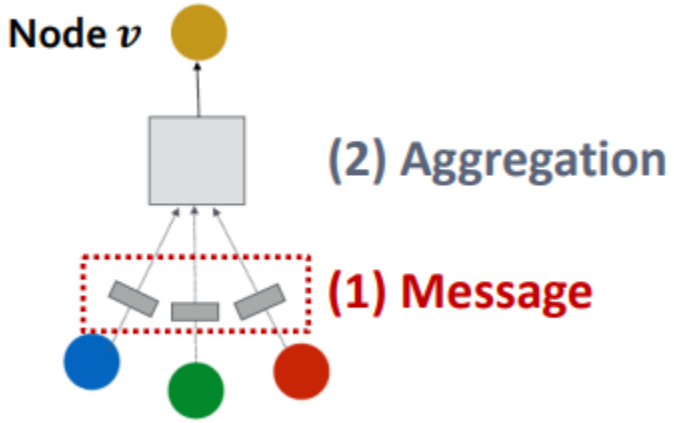
Kipf & Welling (ICLR 2017), related previous works by Duvenaud et al. (NIPS 2015) and Li et al. (ICLR 2016)

Recap: Message Computation

- Intuition: Each node will create a message, which will be sent to other nodes later
- Example: A Linear layer $\mathbf{m}_u^{(l)} = \mathbf{W}^{(l)}\mathbf{h}_u^{(l-1)}$
  - Multiply node features with weight matrix $\mathbf{W}^{(l)}$

Message function: $\mathbf{m}_u^{(l)} = \mathrm{MSG}^{(l)}\left(\mathbf{h}_u^{(l-1)}\right)$





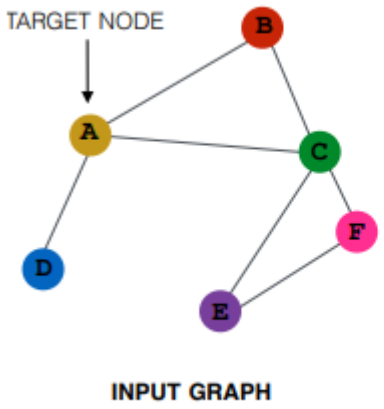
Recap: Message Aggregation

- **Intuition**: Each node will aggregate the messages from node v's neighbors

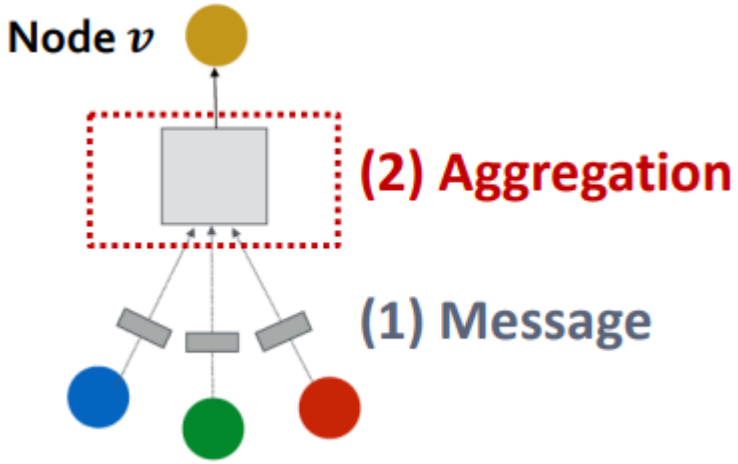
$$\mathbf{h}_v^{(l)} = \mathrm{AGG}^{(l)}\left(\left\{\mathbf{m}_u^{(l)}, u \in N(v)\right\}\right)$$

- **Example**: Sum(·), Mean(·) or Max(·) aggregator

$$\mathbf{h}_v^{(l)} = \mathrm{Sum}(\{\mathbf{m}_u^{(l)}, u \in N(v)\})$$

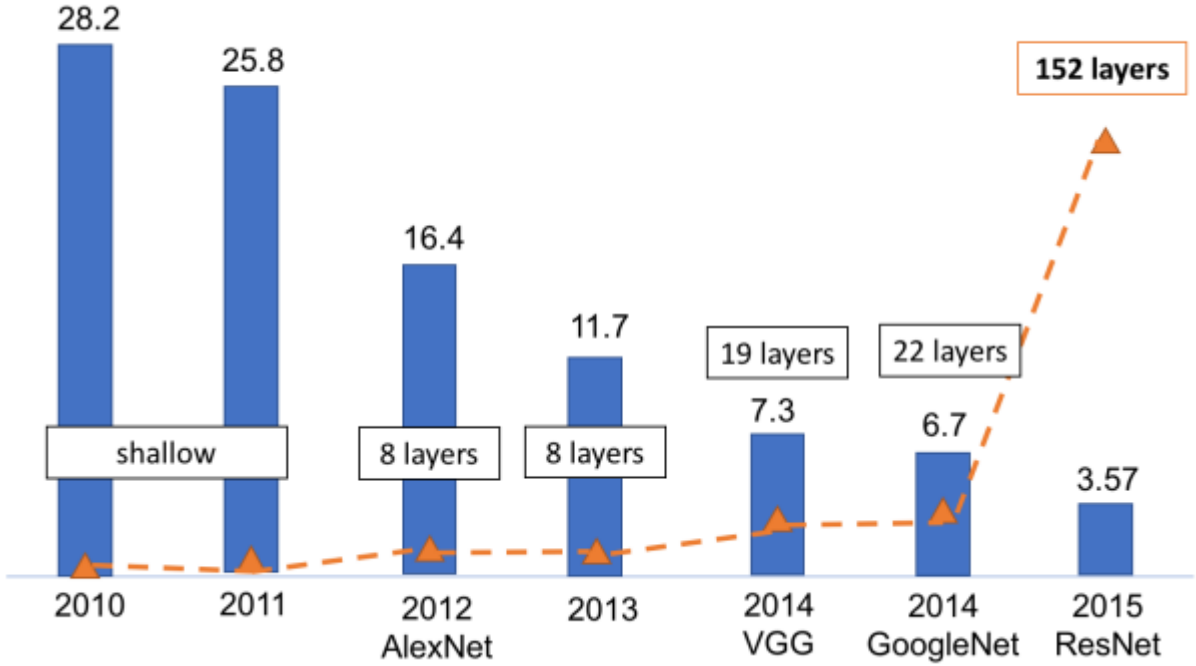


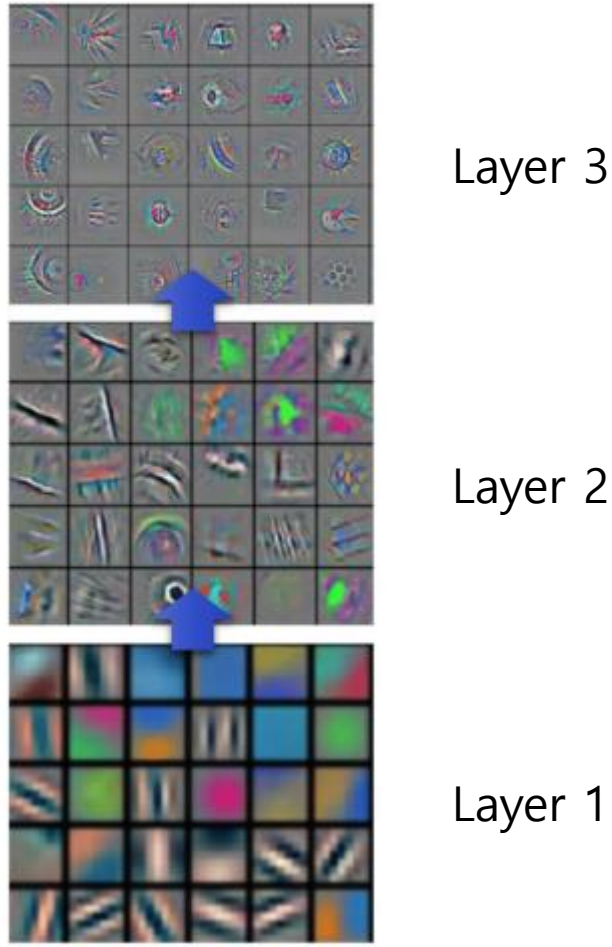
INPUT GRAPH



The Power of Deeper DNNs

- Unprecedented success of deep DNNs in computer vision
- Deeper DNNs enable larger receptive fields

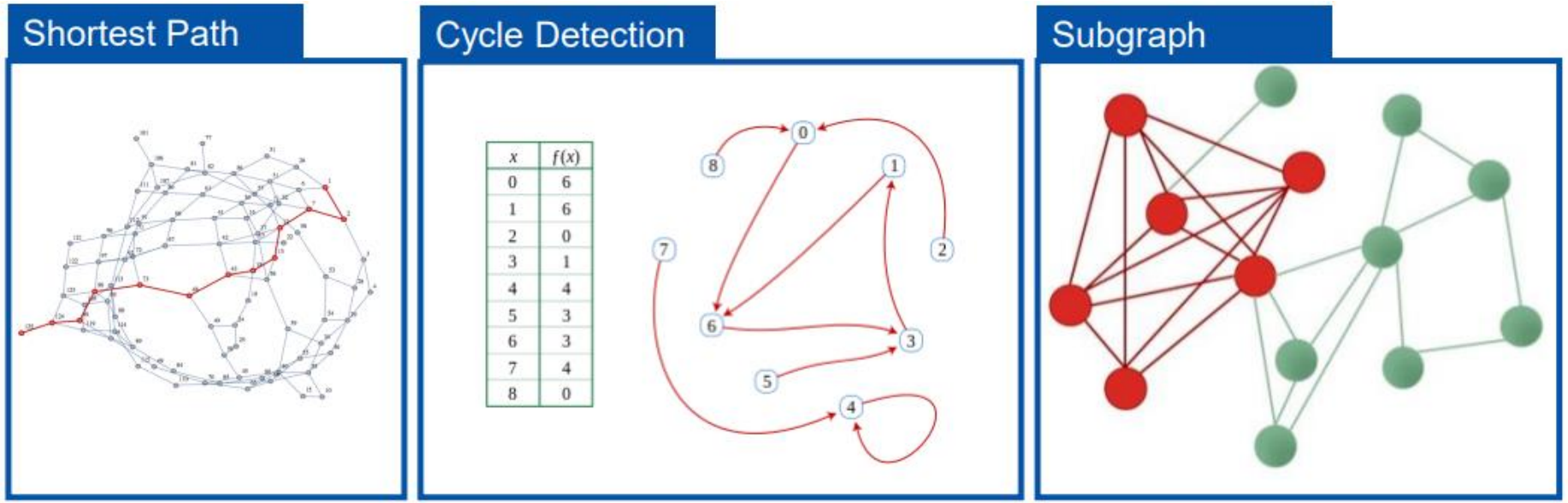




Yu Rong, Wenbing Huang, Tingyang Xu, Hong Cheng, Junzhou Huang 2020

The Power of Deeper DNNs

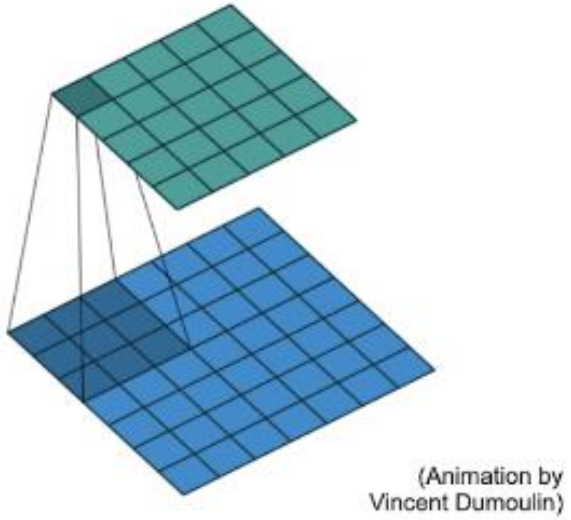
- Do GNNs need deeper structures to enable larger receptive fields, too?
- What limits the expressive power of GNNs?
  - The depth d
  - The width w
- GNNs significantly lose their power when capacity, dw, is restricted



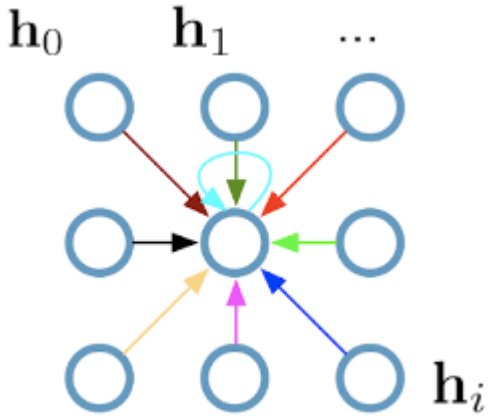
Loukas, Andreas. "What graph neural networks cannot learn: depth vs width." International Conference on Learning Representations. 2020

Recap: CNN vs. GCN

- Single CNN layer with 3x3 filter:



(Animation by Vincent Dumoulin)



Update for a single pixel:

- Transform messages individually $\mathbf{W}_i\mathbf{h}_i$
- Add everything up $\sum_i \mathbf{W}_i\mathbf{h}_i$

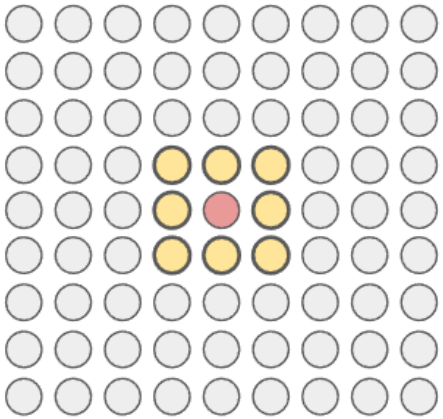
$\mathbf{h}_i \in \mathbb{R}^F$ are (hidden layer) activations of a pixel/node

Full update:

$$\mathbf{h}_4^{(l+1)} = \sigma\left(\mathbf{W}_0^{(l)}\mathbf{h}_0^{(l)} + \mathbf{W}_1^{(l)}\mathbf{h}_1^{(l)} + \cdots + \mathbf{W}_8^{(l)}\mathbf{h}_8^{(l)}\right)$$

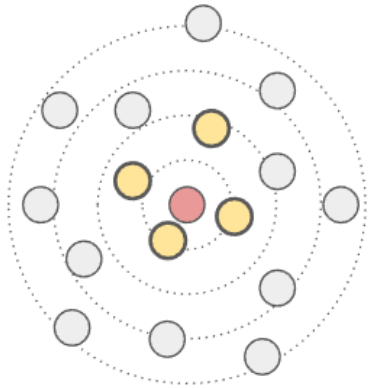
Recap: CNN vs. GCN

- The key differences between CNNs and GCNs is how the convolution is computed.
- For CNNs, we simply do a convolution with our neighbors on the grid with a given kernel size.
- On graphs, we need to define our neighbors first, and then do the convolution in the neighborhood. This is done by finding the k nearest neighbors (with respect to some distance metric) .
- **So, where are the deep GCNs?**



$$\sigma\left(\sum_i w_i x_i + b\right)$$

Convolutional Neural Network (CNN)



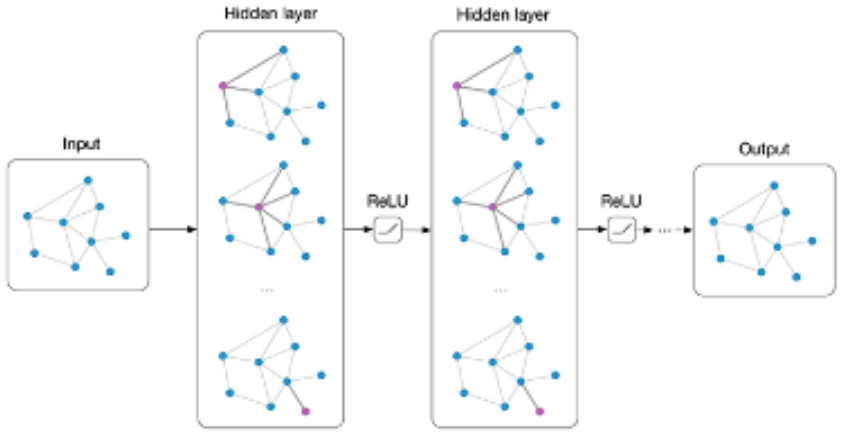
$$\sigma\left(\sum_{u \in \mathcal{N}(v)} w x_u + w' x_v + b\right)$$

Graph Convolutional Network (GCN)

Why current GCNs can not go deeper?

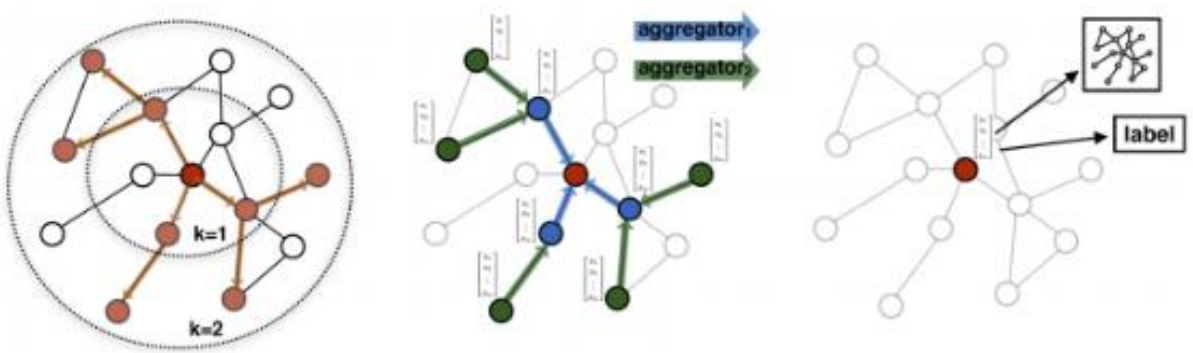
- Most SOTA GCN models are normally no deeper than 3 or 4 layers.

GCN



Kipf, T.N. and Welling, M., 2016. Semi-Supervised Classification with Graph Convolutional Networks.

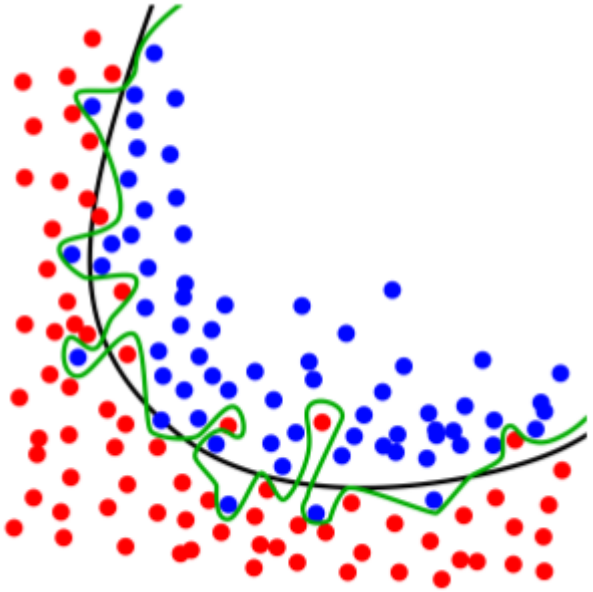
GraphSage



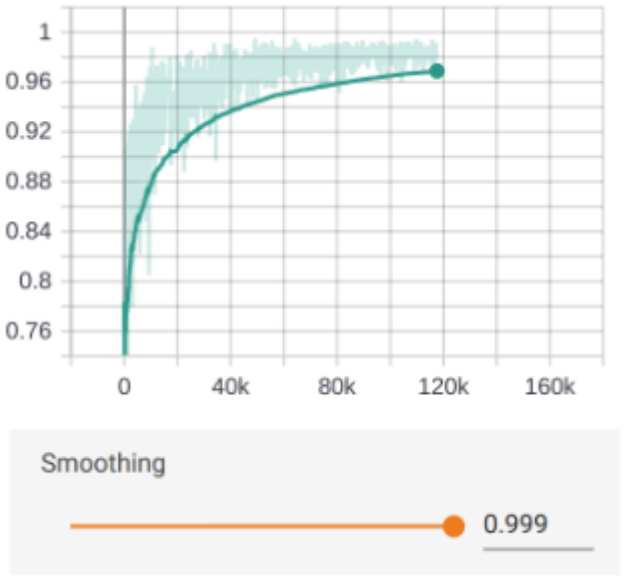
Hamilton, W.L., Ying, R. and Leskovec, J., 2017. Inductive Representation Learning on Large Graphs.

Why GCNs are limited to shallow structures?

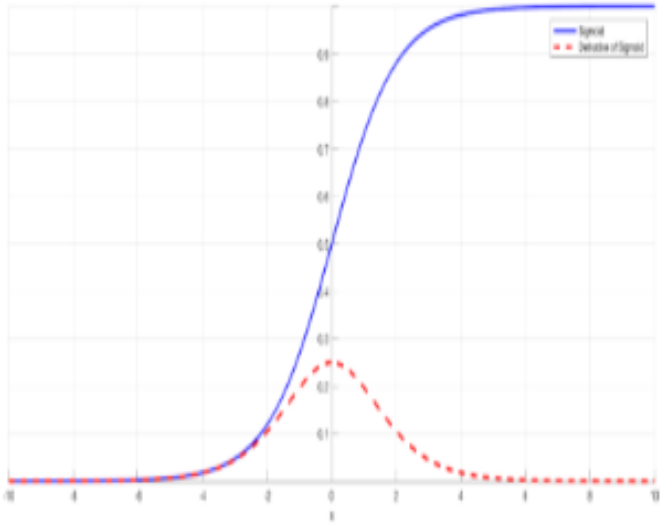
- It turns out that this is not trivial.
- The main difficulties here are Over-fitting, Over-smoothing, and Vanishing gradient.



Over-fitting



Over-smoothing



Vanishing Gradient

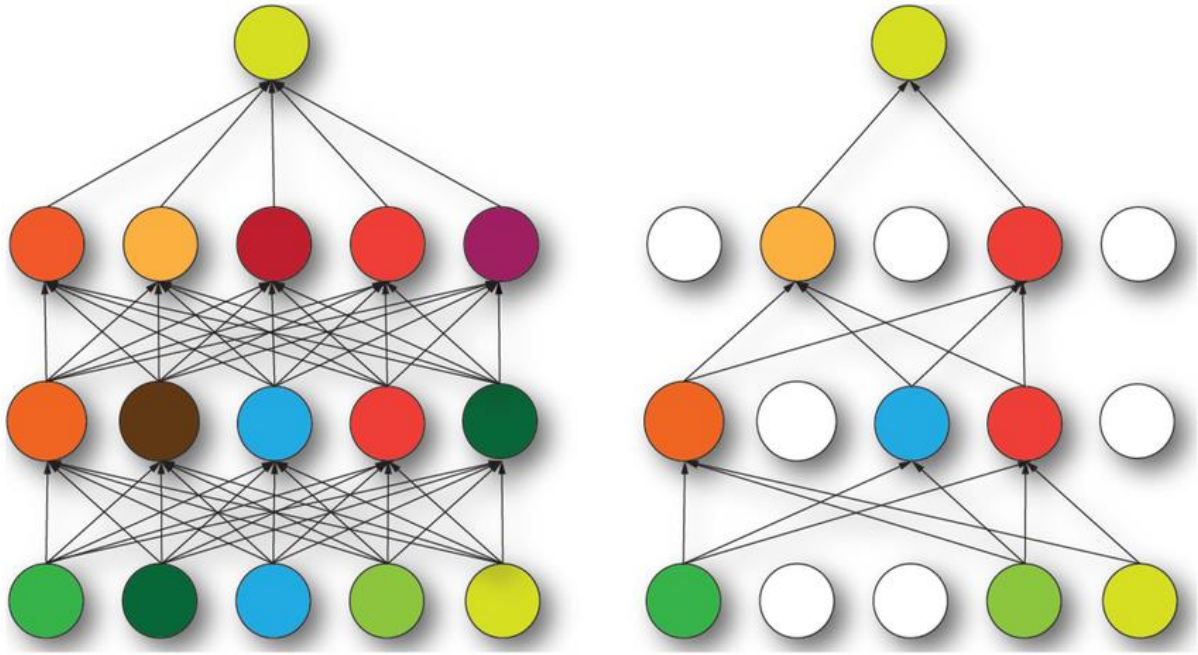
Figures from https://towardsdatascience.com/the-vanishing-gradient-problem-69bf08b15484

Solutions for training Deeper GNNs

- Randomly Dropping
  - Dropout()
  - DropEdges and DropNodes
- JK-Net networks with Skip Connection
- GCNII with Initial residual and Identity mapping
- DeeperGCN with residual connection

Randomly Dropping: Dropout()

- **Similar to standard neural networks:** Dropout is used as a regularization technique — it prevents overfitting by ensuring that no units are codependent (more on this later).
- During training, randomly zeroes some of the elements of the input tensor with probability p using samples from a Bernoulli distribution.



N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: a simple way to prevent neural networks from overfitting

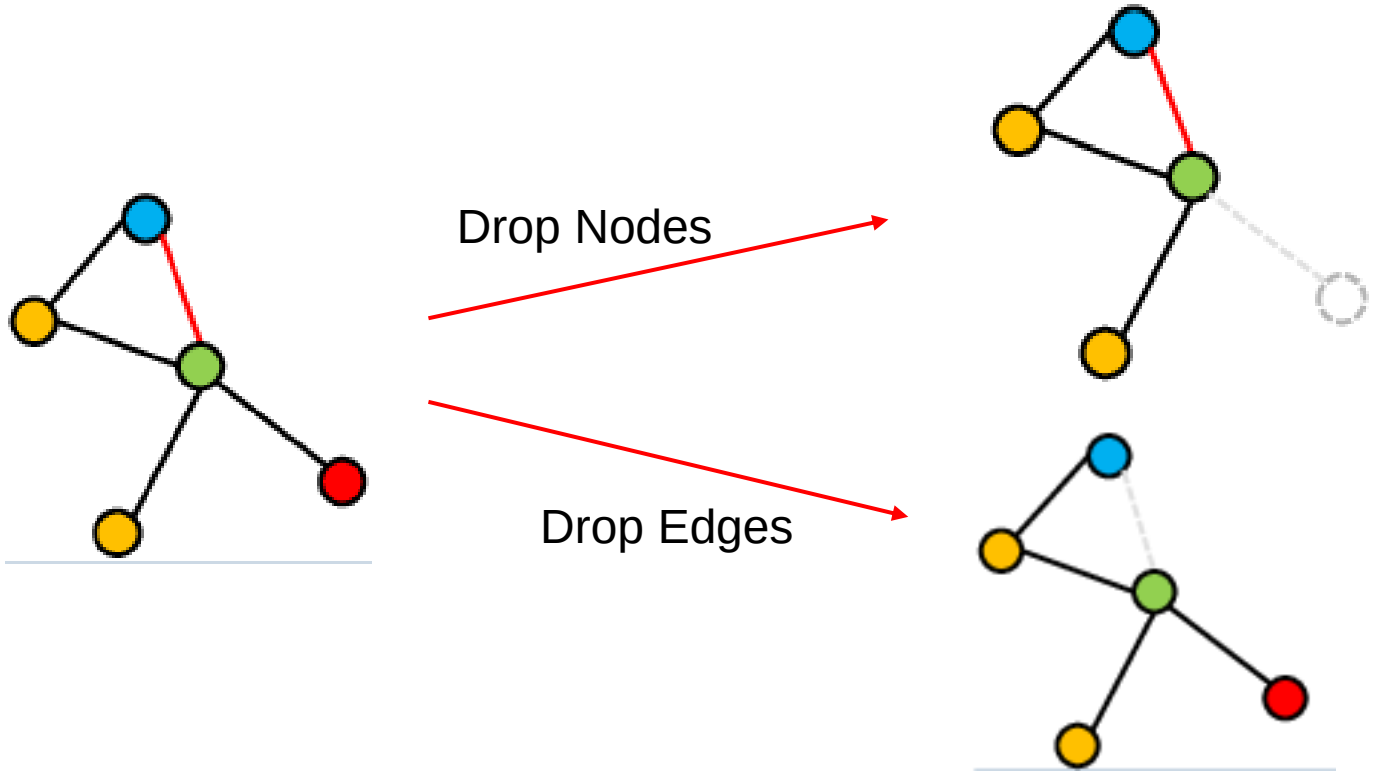
Randomly Dropping: Dropout() Sample code

- **Dropout: From torch.nn.Dropout**
  - During training, randomly zeroes some of the elements of the input tensor with probability p using samples from a Bernoulli distribution.
  - Each channel will be zeroed out independently on every forward call

```
36 class GCN(torch.nn.Module):
37     def __init__(self, num_features, num_classes, dim=16, drop=0.5):
38         super(GCN, self).__init__()
39         self.conv1 = GCNConv(num_features, dim)
40         self.conv2 = GCNConv(dim, num_classes)
41         self.drop = torch.nn.Dropout(p=drop)
42         self.h = None
43     def forward(self, x, edge_index,  g, Kindices):
44         x = F.relu(self.conv1(x, edge_index))
45         x = self.drop(x)
46         x = self.conv2(x, edge_index)
47         self.h = x
48         return F.log_softmax(x, dim=1)
49
```

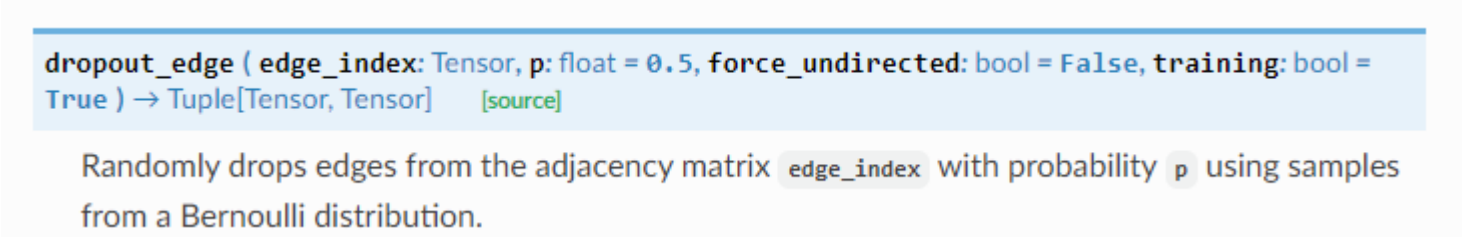
Dropping: Edges and Nodes

- Randomly remove a certain number of edges or nodes from the input graph at each training epoch.
- Alternatively, they could be regarded as data augmentation methods, helping relieve both the over-fitting and oversmoothing issues in training very deep GNNs.



Dropping Edges/Nodes: Sample code

- **Dropout: From torch.nn.Dropout**
  - Randomly drops edges from the adjacency matrix edge_index with probability p using samples from a Bernoulli distribution.



dropout_edge (**edge_index**: Tensor, **p**: float = `0.5`, **force_undirected**: bool = `False`, **training**: bool = `True`) → Tuple[Tensor, Tensor] [source]

Randomly drops edges from the adjacency matrix `edge_index` with probability `p` using samples from a Bernoulli distribution.

Random Dropping: DropEdges Sample code

- Randomly drops edges/nodes for each epochs
- Lets do some sample codes

In [43]:

```python
from torch_geometric.utils import dropout_adj

def train_dropEdges(model, data):
    """Train a GNN model and return the trained model."""
    criterion = torch.nn.CrossEntropyLoss()
    optimizer = model.optimizer
    epochs = 5

    model.train()
    for epoch in range(epochs+1):
        # Training
        optimizer.zero_grad()
        edge_index1, _ = dropout_adj(data.edge_index, data.edge_attr, p=0.5)
        _, out = model(data.x, edge_index1 )
        loss = criterion(out[data.train_mask], data.y[data.train_mask])
        acc = accuracy(out[data.train_mask].argmax(dim=1), data.y[data.train_mask])
        loss.backward()
        optimizer.step()

        # Validation
        val_loss = criterion(out[data.val_mask], data.y[data.val_mask])
        val_acc = accuracy(out[data.val_mask].argmax(dim=1), data.y[data.val_mask])

        # Print metrics every 10 epochs
        if(epoch % 1 == 0):
            print(f'Epoch {epoch:>3} | Train Loss: {loss:.3f} | Train Acc: '
                  f'{acc*100:>6.2f}% | Val Loss: {val_loss:.2f} | '
                  f'Val Acc: {val_acc*100:.2f}%')

    return model
```

```python
from torch_geometric.utils import dropout_node

def train_dropNodes(model, data):
    """Train a GNN model and return the trained model."""
    criterion = torch.nn.CrossEntropyLoss()
    optimizer = model.optimizer
    epochs = 5

    model.train()
    for epoch in range(epochs+1):
        # Training
        optimizer.zero_grad()
        edge_index1, _, _ = dropout_node(data.edge_index)
        _, out = model(data.x, edge_index1 )
        loss = criterion(out[data.train_mask], data.y[data.train_mask])
        acc = accuracy(out[data.train_mask].argmax(dim=1), data.y[data.train_mask])
        loss.backward()
        optimizer.step()

        # Validation
        val_loss = criterion(out[data.val_mask], data.y[data.val_mask])
        val_acc = accuracy(out[data.val_mask].argmax(dim=1), data.y[data.val_mask])

        # Print metrics every 10 epochs
        if(epoch % 1 == 0):
            print(f'Epoch {epoch:>3} | Train Loss: {loss:.3f} | Train Acc: '
                  f'{acc*100:>6.2f}% | Val Loss: {val_loss:.2f} | '
                  f'Val Acc: {val_acc*100:.2f}%')

    return model
```

JK-Net networks

- **Idea:**
  1) Makes each layer increase the size of the influence distribution by aggregating neighborhoods from the previous layers
  2) Combines some of the previous layers' representations independently for each node

- At the last layer, for each node, carefully select from all of those itermediate representations (which "jump" to the last layer).
- If this is done independently for each node, then the model can adapt the effective neighborhood size for each node as needed, resulting in exactly the desired adaptivity.

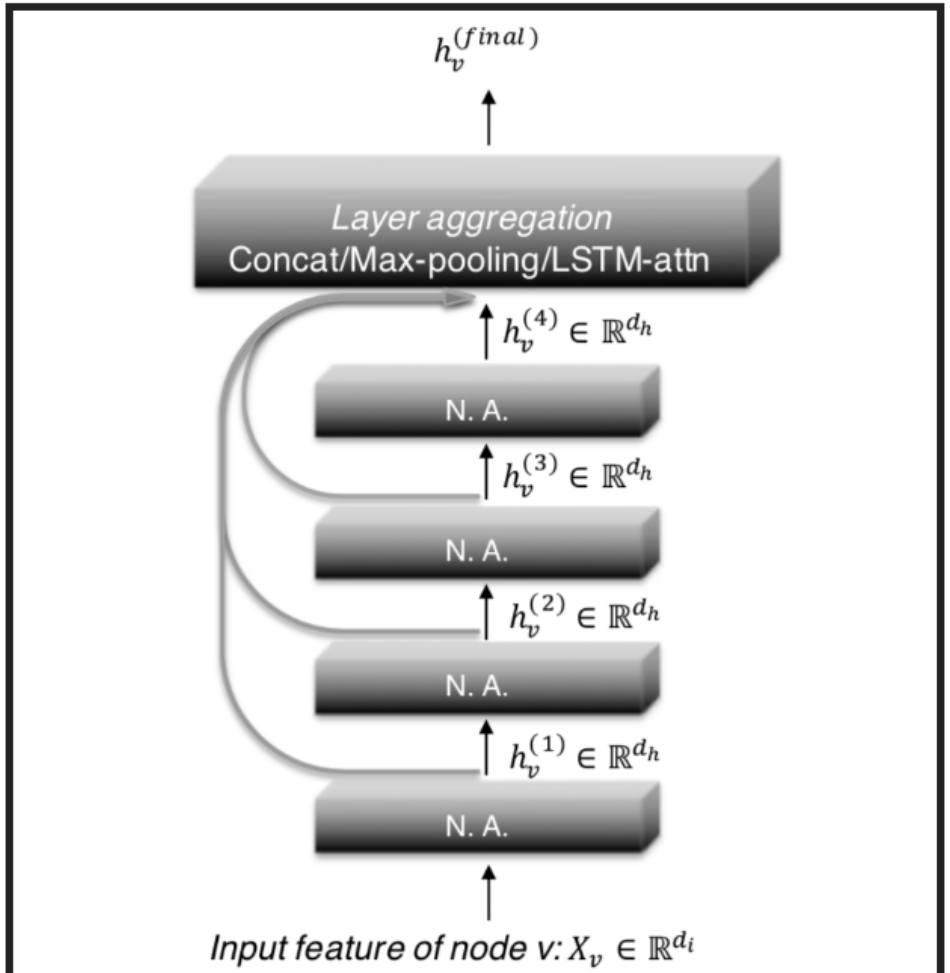


Figure 4. A 4-layer Jumping Knowledge Network (JK-Net). N.A. stands for neighborhood aggregation.

JK networks: Aggregation Mechanisms with Skip Connections

- **CONCATENATION**:

$$\text{CONCATENATION}\left[h_v^{(1)}, \ldots, h_v^{(k)}\right]$$

- **MAX-POOLING**: Select the most informative layer for each feature coordinate.
- **LSTM-ATTENTION**: Input $h_v^{(1)}, \ldots, h_v^{(k)}$ into a bi-directional LSTM to generate forward and backward features $f_v^{(l)}$ and $b_v^{(l)}$ for each layer

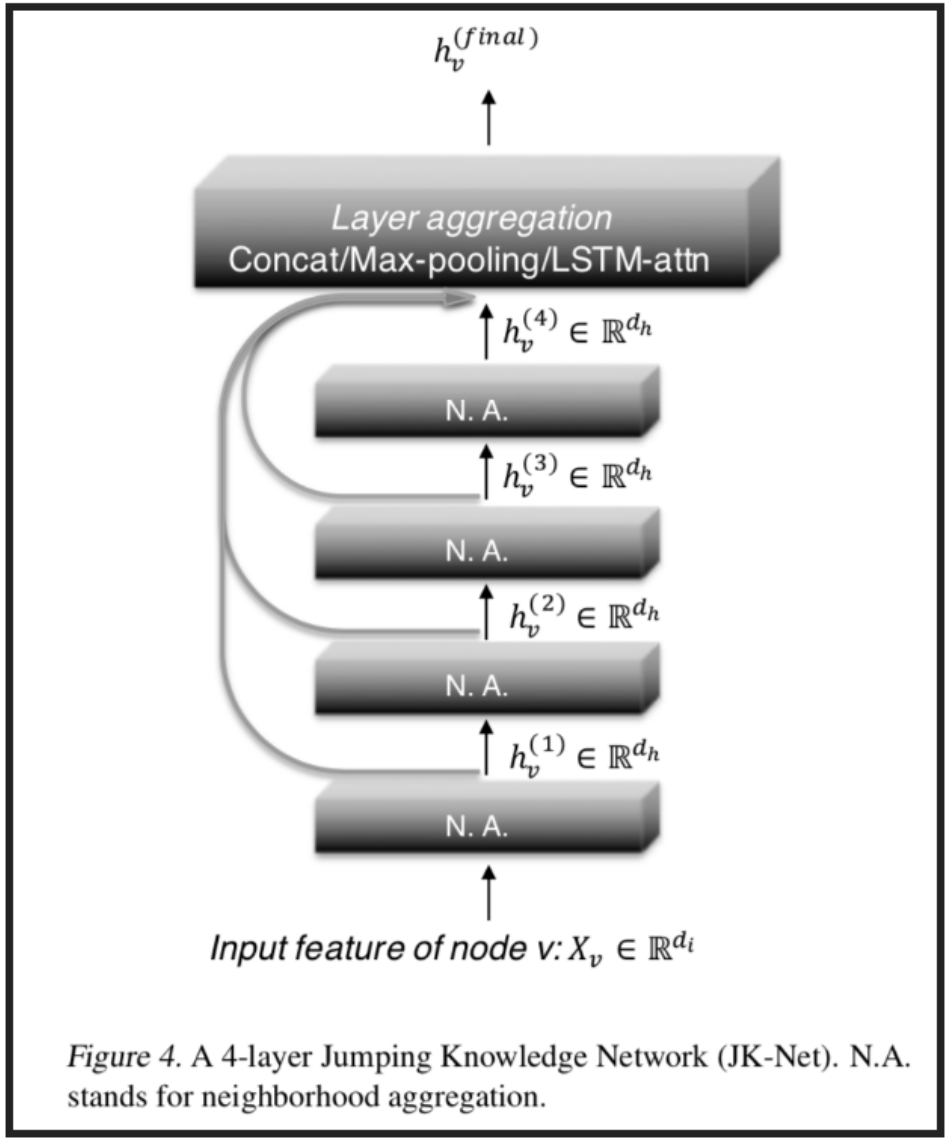
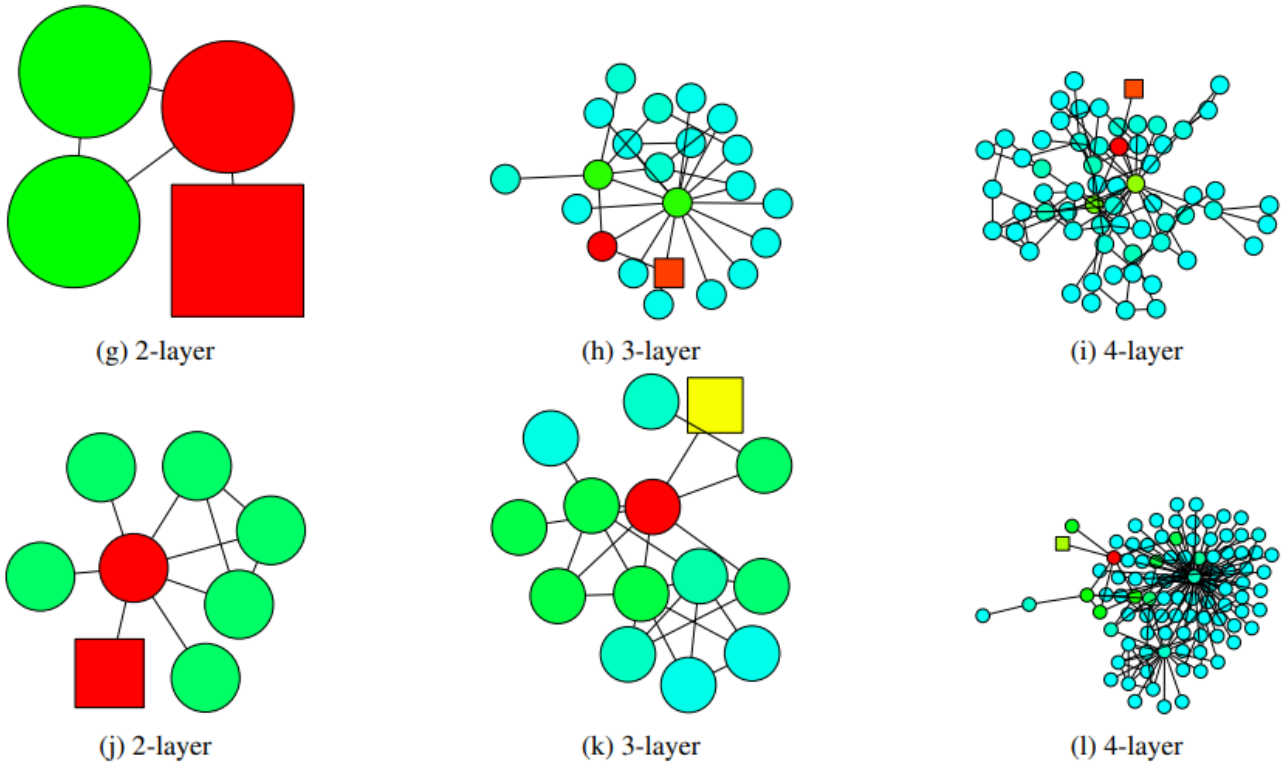


Figure 4. A 4-layer Jumping Knowledge Network (JK-Net). N.A. stands for neighborhood aggregation.

JK-Net networks: Example

- A 6-layer JK-Net learns to adapt to different subgraph structures



JK-Net networks: Sample code

JK-Net in PyTorch Geometric:

```
from torch_geometric.nn import JumpingKnowledge
```

torch_geometric.nn / models.JumpingKnowledge

models.JumpingKnowledge

class **JumpingKnowledge** (**mode**: str, **channels**: Optional[int] = **None**, **num_layers**: Optional[int] = **None**)
[source]

Bases: `Module`

The Jumping Knowledge layer aggregation module from the "Representation Learning on Graphs with Jumping Knowledge Networks" paper based on either **concatenation** (`"cat"`)

$$\mathbf{x}_v^{(1)} \, \| \, \ldots \, \| \, \mathbf{x}_v^{(T)}$$

max pooling (`"max"`)

$$\max\left(\mathbf{x}_v^{(1)}, \ldots, \mathbf{x}_v^{(T)}\right)$$

or **weighted summation**

$$\sum_{t=1}^{T} \alpha_v^{(t)} \mathbf{x}_v^{(t)}$$

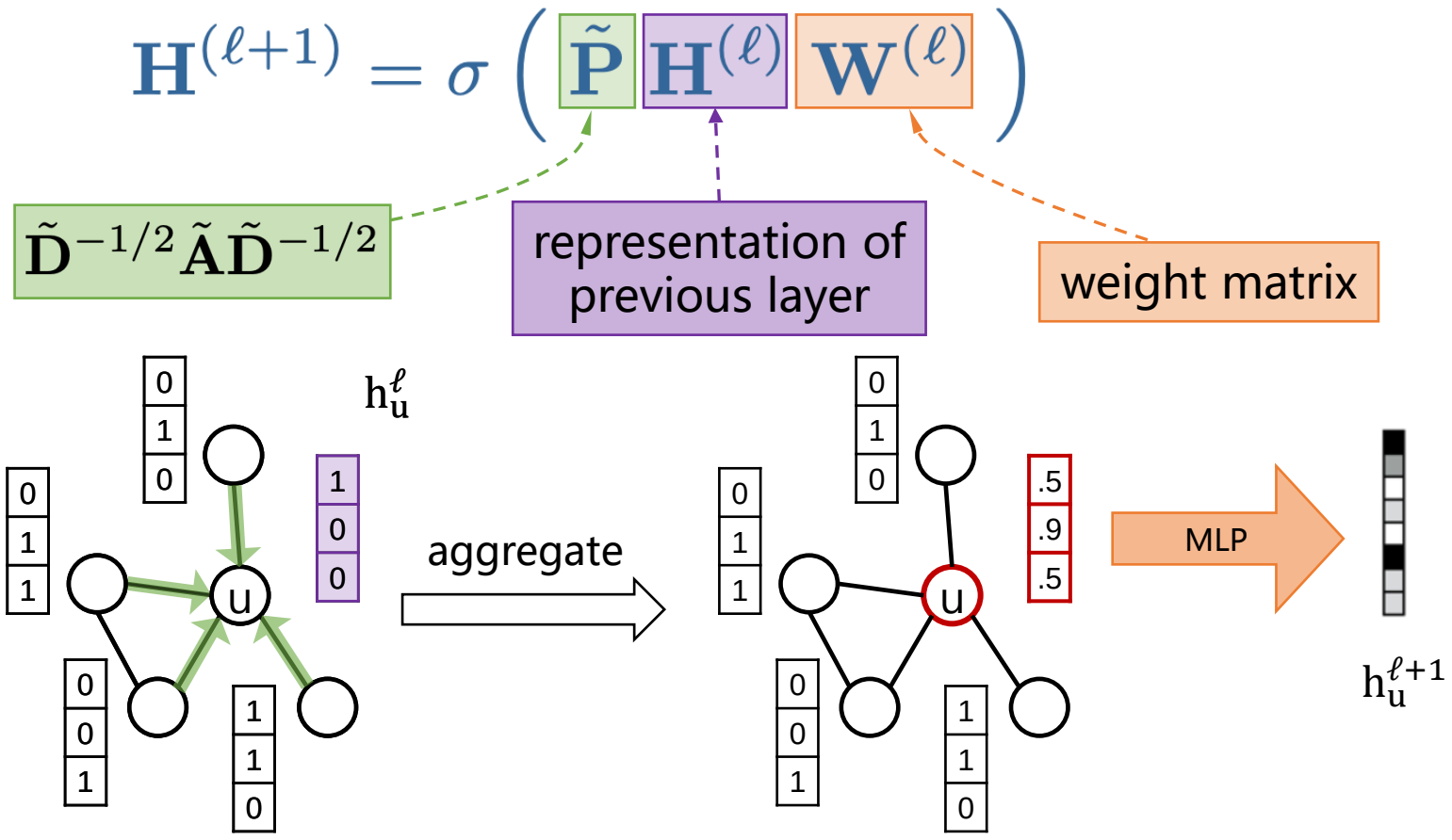
https://pytorch-geometric.readthedocs.io/en/latest/generated/torch_geometric.nn.models.JumpingKnowledge.html

JK-Net networks: Sample code

- Lets do some sample codes

```
class JKNet(torch.nn.Module):
    def __init__(self, dataset, mode='max', num_layers=6, hidden=16):
        super(JKNet, self).__init__()
        self.num_layers = num_layers
        self.mode = mode

        self.conv0 = GCNConv(dataset.num_node_features, hidden)
        self.dropout0 = torch.nn.Dropout(p=0.5)

        for i in range(1, self.num_layers):
            setattr(self, 'conv{}'.format(i), GCNConv(hidden, hidden))
            setattr(self, 'dropout{}'.format(i), torch.nn.Dropout(p=0.5))

        self.jk = JumpingKnowledge(mode=mode)
        if mode == 'max':
            self.fc = torch.nn.Linear(hidden, dataset.num_classes)
        elif mode == 'cat':
            self.fc = torch.nn.Linear(num_layers * hidden, dataset.num_classes)
        self.optimizer = torch.optim.Adam(self.parameters(),
                                          lr=0.01,
                                          weight_decay=5e-4)
    def forward(self, x, edge_index):

        layer_out = []
        for i in range(self.num_layers):
            conv = getattr(self, 'conv{}'.format(i))
            dropout = getattr(self, 'dropout{}'.format(i))
            x = dropout(F.relu(conv(x, edge_index)))
            layer_out.append(x)

        h = self.jk(layer_out)
        h = self.fc(h)
        return h, F.log_softmax(h, dim=1)
```

- Graph Convolutional Layer (Kipf et al., 2017)



GCNII: GCN via Initial residual and Identity mapping

- Vanilla GCN

$$\mathbf{H}^{(\ell+1)} = \sigma\left(\tilde{\mathbf{P}}\mathbf{H}^{(\ell)}\mathbf{W}^{(\ell)}\right)$$

- GCN + **Initial residual**

$$\mathbf{H}^{(\ell+1)} = \sigma\left(\left((1-\alpha_\ell)\tilde{\mathbf{P}}\mathbf{H}^{(\ell)} + \alpha_\ell \mathbf{H}^{(0)}\right)\mathbf{W}^{(\ell)}\right)$$

- **GCNII**: GCN + Initial residual **+ Identity mapping**

$$\mathbf{H}^{(\ell+1)} = \sigma\left(\left((1-\alpha_\ell)\tilde{\mathbf{P}}\mathbf{H}^{(\ell)} + \alpha_\ell \mathbf{H}^{(0)}\right)\left((1-\beta_\ell)\mathbf{I}_n + \beta_\ell \mathbf{W}^{(\ell)}\right)\right)$$

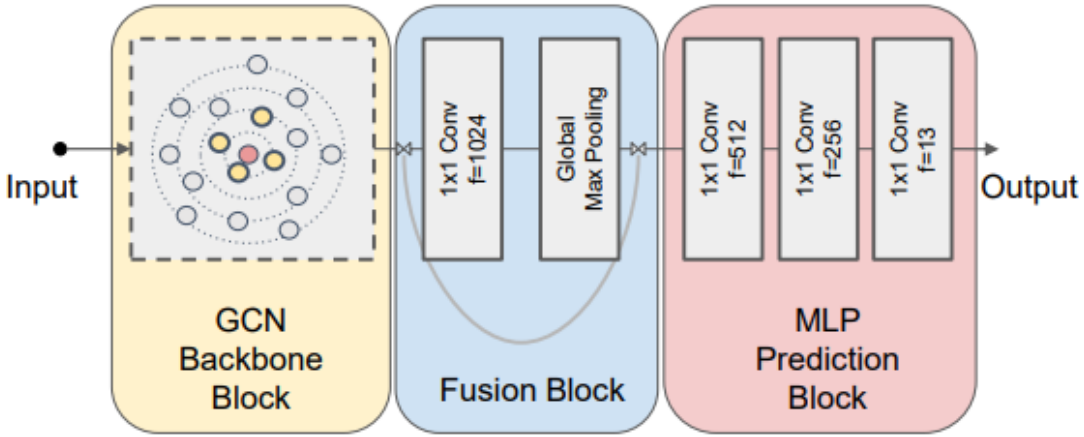
GCNII: Sample code

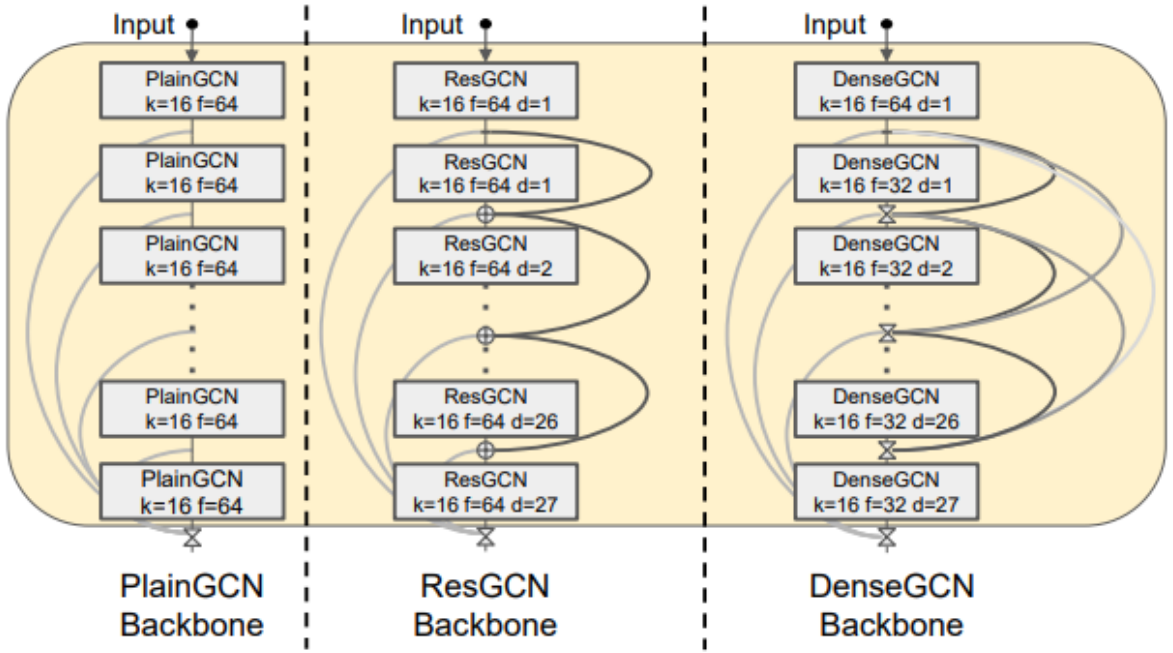
from torch_geometric.nn import GCN2Conv

```
19  class Net(torch.nn.Module):
20      def __init__(self, hidden_channels, num_layers, alpha, theta,
21                   shared_weights=True, dropout=0.0):
22          super().__init__()
23
24          self.lins = torch.nn.ModuleList()
25          self.lins.append(Linear(dataset.num_features, hidden_channels))
26          self.lins.append(Linear(hidden_channels, dataset.num_classes))
27
28          self.convs = torch.nn.ModuleList()
29          for layer in range(num_layers):
30              self.convs.append(
31                  GCN2Conv(hidden_channels, alpha, theta, layer + 1,
32                           shared_weights, normalize=False))
33
34          self.dropout = dropout
35
```

DeepGCNs framework

- Solving vanishing gradient problem by using Residual Learning for GCNs





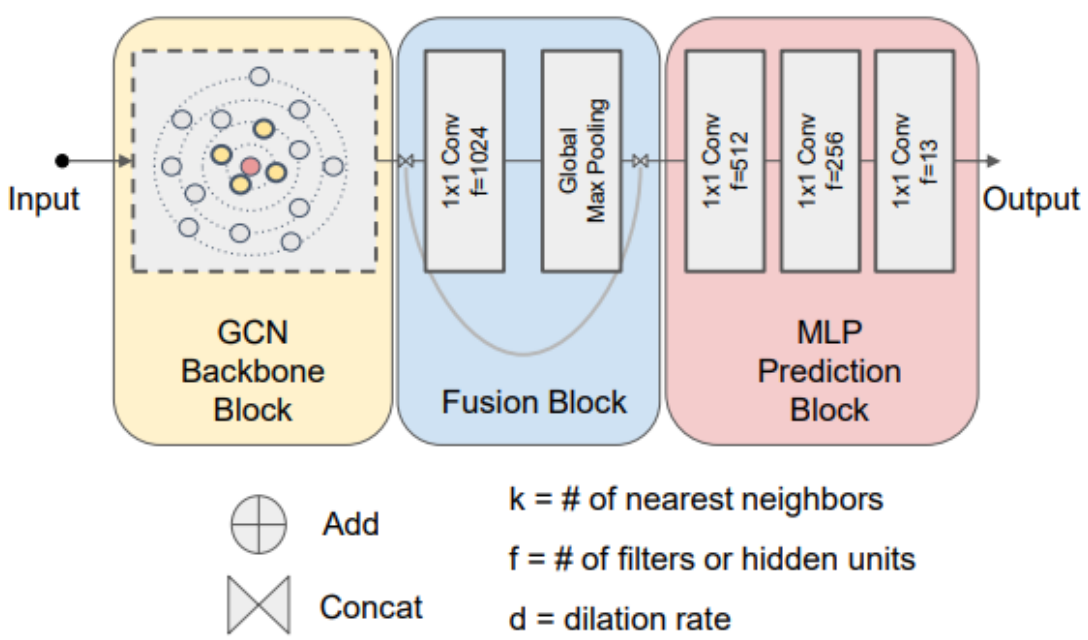
$$\begin{aligned}\mathcal{G}_{l+1} &= \mathcal{H}(\mathcal{G}_l, \mathcal{W}_l) \\ &= \mathcal{F}(\mathcal{G}_l, \mathcal{W}_l) + \mathcal{G}_l = \mathcal{G}^{res}_{l+1} + \mathcal{G}_l.\end{aligned} \tag{3}$$

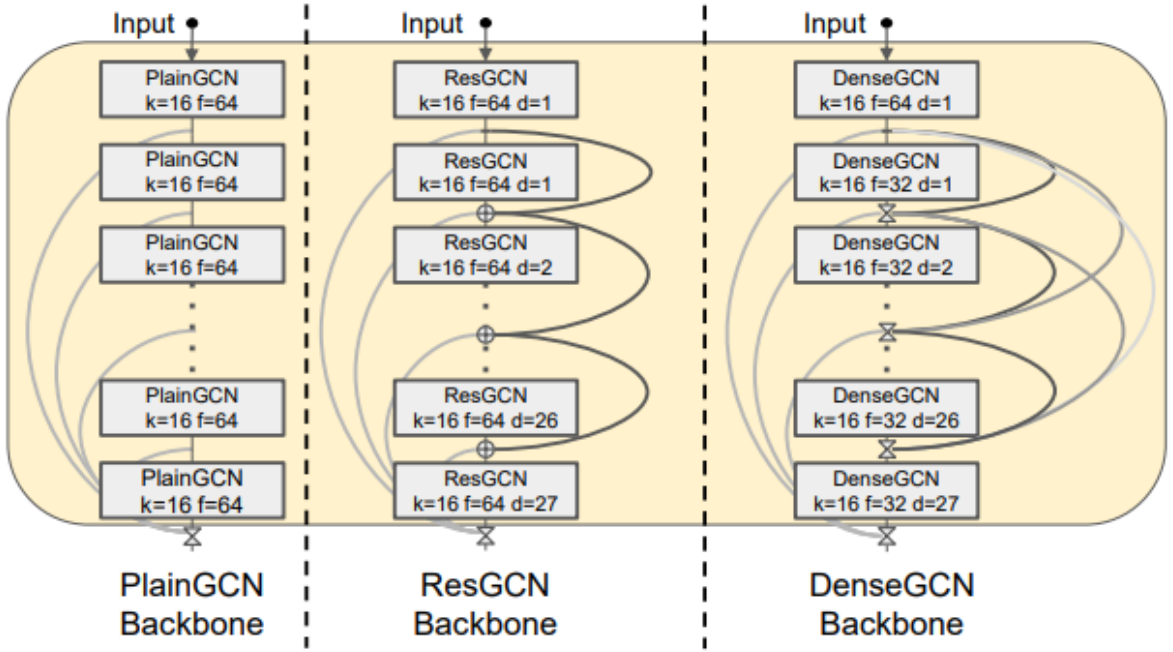
$\mathcal{G}_l = (\mathcal{V}_l, \mathcal{E}_l)$ and $\mathcal{G}_{l+1} = (\mathcal{V}_{l+1}, \mathcal{E}_{l+1})$ are the input and output graphs at the l-th layer

DeepGCNs: Can GCNs Go as Deep as CNNs?; ICCV 2019

DeepGCNs framework

- Framework consists of three blocks (GCN Backbone Block, Fusion Block and MLP Prediction Block).
- Three types of GCN Backbone Blocks (PlainGCN, ResGCN and DenseGCN)
- There are two kinds of GCN skip connections vertex-wise additions and vertex-wise concatenations. k is the number of nearest neighbors in GCN layers

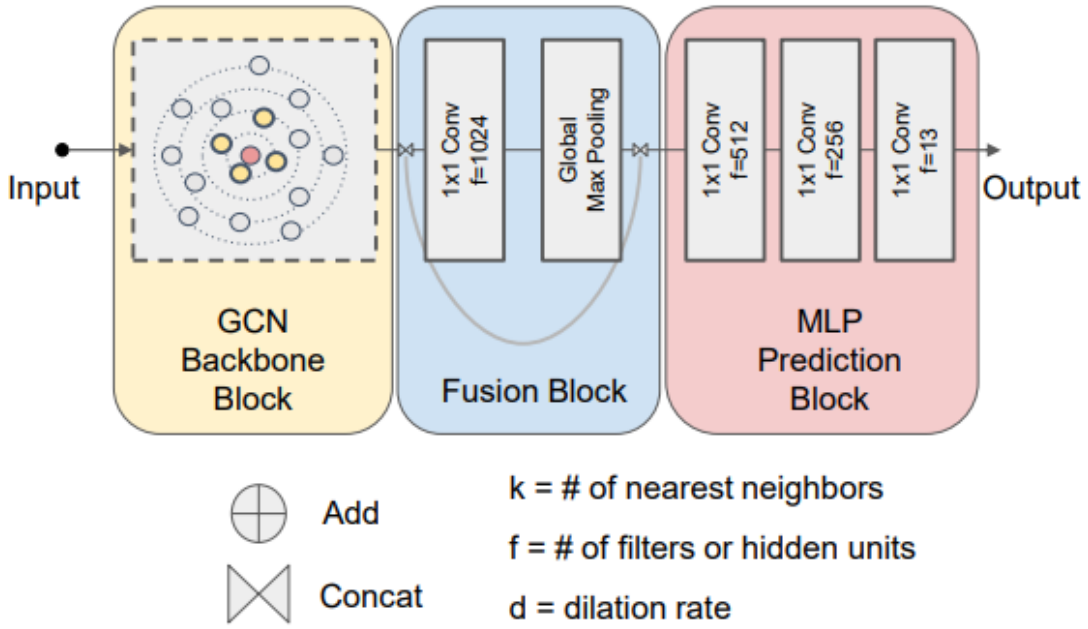


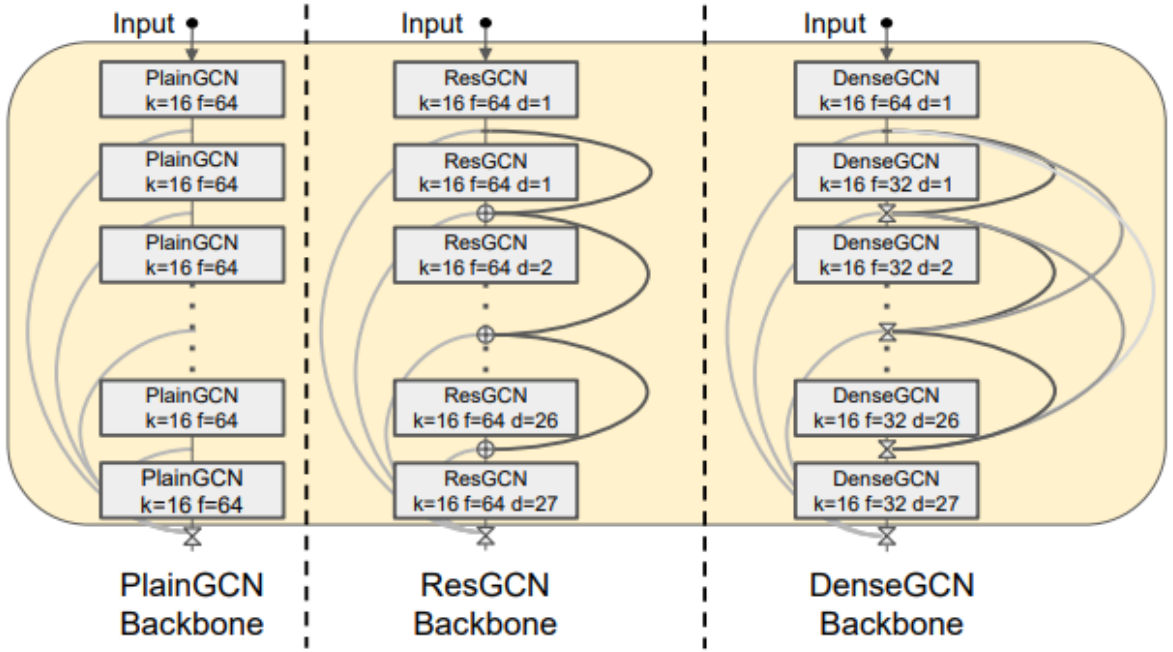


DeepGCNs: Can GCNs Go as Deep as CNNs?; ICCV 2019

DeepGCNs framework

- **PlainGCN**: a fusion block, and a MLP prediction block
- **ResGCN**: construct ResGCN by adding dynamic dilated k-NN and residual graph connections to PlainGCN.
- **DenseGCN**: adding dynamic dilated k-NN and dense graph connections to the PlainGCN.





DeepGCNs: Can GCNs Go as Deep as CNNs?; ICCV 2019

DeepGCNs: Sample code

- Lets do some sample codes

```
class DeeperGCN(torch.nn.Module):
    def __init__(self,num_features, num_classes, dim =16, num_layers =2, drop=0.5):
        super().__init__()

        self.node_encoder = Linear(num_features, dim)
        #self.edge_encoder = Linear(data.edge_attr.size(-1), hidden_channels)
        self.h = None
        self.layers = torch.nn.ModuleList()
        for i in range(1, num_layers + 1):
            conv = GENConv(dim, dim, aggr='softmax', t=1.0, learn_t=True, num_layers=2, norm='layer')
            norm = LayerNorm(dim, elementwise_affine=True)
            act = ReLU(inplace=True)

            layer = DeepGCNLayer(conv, norm, act, block='res+', dropout=0.1,ckpt_grad=i % 3)
            self.layers.append(layer)
        self.mlp = Linear(dim, num_classes)
        self.optimizer = torch.optim.Adam(self.parameters(),lr=0.01,weight_decay=5e-4)


    def forward(self, x, edge_index):
        x = self.node_encoder(x)
        x = self.layers[0].conv(x, edge_index)

        for layer in self.layers[1:]:
            x = layer(x, edge_index)

        x = self.layers[0].act(self.layers[0].norm(x))
        x = F.dropout(x, p=0.1, training=self.training)
        x = self.mlp(x)
        self.h = x
        return x, F.log_softmax(x, dim=1)
```

네트워크 과학 연구실
NETWORK SCIENCE LAB

가톨릭대학교
THE CATHOLIC UNIVERSITY OF KOREA